2019 IEEE/ACM 7th International Workshop on Software Engineering for Systems-of-Systems (SESoS) and 13th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES)

Slicing Executable System-of-Systems Models for Efficient Statistical Verification

Jiyoung Song, Jacob O. Tørring, Sangwon Hyun, Eunkyoung Jee, Doo-Hwan Bae
School of Computing
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Republic of Korea
{jysong, jacobot, swhyun, ekjee, bae}@se.kaist.ac.kr

***Abstract*—A System of Systems (SoS), composed of independent constituent systems, can create synergy among its systems to achieve a common goal. Many studies have used statistical model checking techniques to verify how well an SoS can achieve its goals. SoS models are usually complex and probabilistic, which makes statistical verification computationally expensive. To reduce this cost, dynamic slicing techniques can be applied to SoS models since both dynamic slicing and statistical verification focus on the models' execution samples. However, existing dynamic slicing techniques cannot guarantee executable accurate slices of SoS models when the models contain uncertainty. Therefore, we propose a hybrid slicing approach that combines dynamic backward slicing and modified observation-based slicing to produce accurate executable slices. Experimentation on the proposed technique found that the verification time was significantly reduced (47-56%), depending on the property, while preserving the verification results.**

***Index Terms*—System-of-Systems, model slicing, statistical model checking, model verification**

I. Introduction

A System of Systems (SoS) consists of autonomous and independent Constituent Systems (CSs) that can achieve common goals. It is vital to verify that defects have not occurred during the process of integrating independent CSs and that the orchestrated CSs are sufficient to achieve the desired goal(s). SoS managers can perform model-based verification by abstracting large systems into manageable models and verifying the models with respect to the desired properties, such as goal achievement and safety. In previous works [1]–[4], the authors modeled and verified an SoS using Statistical Model Checking (SMC) techniques. Although SMC techniques are much more scalable than classical exhaustive model checking techniques, the cost of SMC is still significant in many SoSs due to the size of the SoS model and the cost of simulations.

To reduce the cost of SoS model verification, slicing techniques can be considered. Song et al [3] proposed SoS GaP slicer—a slicing framework for SoS goal models and simulation models that is based on SoS changes. The changed parts and change-related parts of the SoS model are sliced by backward slicing, and the slices are given to the verification engine. SoS GaP slicer focuses on SoS change analysis, and it only uses verification properties that are related to change as the slicing criteria. However, SoS managers need to analyze SoS model based on other properties related to existence, absence, universality, transient states, steady states, and minimum duration.

Applying dynamic slicing on the probabilistic SoS simulation model can increase the efficiency of statistical verification by reducing size of the model. SMC and dynamic slicing are both performed on actual model executions. SMC verifies the model not by checking every execution path but rather by sampling several executions of the model. Slicing the probabilistic SoS simulation model based on several execution samples reduces the scope of the probabilistic SoS simulation model to be sliced. By contrast, if we perform static slicing on an entire SoS model, the cost of analysis will be substantially larger. However, existing dynamic slicing approaches do not generate executable and accurate slices of probabilistic SoS simulation models. JavaSlicer [5][1], a dynamic slicing tool, cannot generate executable slices because the purpose of the tool is to profile Java programs for parallelism. Another slicing approach, observation-based slicing (ORBS) [6], compares the intended simulation trajectories to the modified simulation trajectories of the subject programs. When applying ORBS directly to the probabilistic SoS simulation models that have uncertainty, we cannot compare the simulation trajectories because they will vary between executions due to the uncertainty.

In this paper, we propose a general analysis approach that slices probabilistic SoS simulation models based on user-interested variables, statements, and verification properties. We utilize dynamic backward slicing for generating accurate slices and the slices become executable by modified observation-based slicing. This hybrid approach complements the limitations of JavaSlicer and ORBS approaches. We have implemented the proposed hybrid approach as a model slicer module of Simulation-based Verification and Analysis (SIMVA) for SoS, an integrated tool for the statistical verification of SoS. SIMVA-SoS has been released with several scenarios such as a Mass Casualty Incident (MCI) response SoS in GitHub[2]. The input of the proposed approach, the probabilistic SoS simulation model is a Java program, whereas the output is an executable probabilistic SoS simulation model slice that enables the efficient use of SMC. In our experiment, we show the accuracy of the proposed approach by comparing

[1]https://github.com/hammacher/javaslicer
[2]https://github.com/SESoS/SIMVA-SoS

verification results of SoS model slices that are generated by our tool to those of the original SoS model. Our experimental results show that the verification time has been reduced by 47-56%, depending on the property, when the SoS model slices are given to the SMC engine, while the verification results are preserved.

The remainder of this paper is organized as follows: section II explains the background of both the statistical model checking and the slicing techniques; section III illustrates an MCI-response SoS scenario and motivating slicing examples; section IV presents the proposed approach and its implementation; in section V, we apply our approach to a probabilistic SoS simulation model and conduct experiments with three research questions; section VI describes related works to our approach; and we discuss our conclusions in section VII.

II. Background

In this section, we firstly describe the mechanism of SMC and the property specification that are used in SIMVA-SoS, and we follow with a brief introduction to slicing techniques.

A. Statistical Model Checking in SIMVA-SoS

The SMC technique deduces whether the system satisfies the property by simulating and monitoring the system via hypothesis testing, that can give statistical evidence for the satisfaction or the violation of the specification [7]. SMC has been used to verify large and complex systems because it is less time and memory intensive than numerical model checking [8]. The SIMVA-SoS tool's SMC engine consists of three modules: simulation, verification, and analysis. In addition, there are three inputs for the engine: the executable model S, the verification property ϕ, and the precision parameters.

The simulation module executes the executable model S, which can represent the stochastic behavior of a target system. The discrete-time Markov chain (DTMC) and the continuous-time Markov chain (CTMC) models can be used to construct the stochastic model. Both models can represent probabilistic transitions between finite states, but the main difference is that the CTMC considers continuous time [9]. Via this executable model, the simulation module sends a simulation trace σ to the verification module. Because simulation traces of a stochastic model are not always the same, a statistical analysis with repetitive simulations is necessary.

The verification module checks whether the simulation trace σ satisfies the verification property ϕ. There are many kinds of temporal logic for formulating verification properties [10]. The SIMVA-SoS verification module can handle probabilistic computational logic (PCTL) for DTMC model verification and continuous stochastic logic (CSL) for CTMC model verification. An example of a PCTL property is as follows: "P =?[trueU <= 10000(num_of_patient_SEA) >= 45]". This example is used in the MCI-response system scenario in SIMVA-SoS, where it checks whether the system finds more than 45 patients at sea within 10,000 steps for each trace. Finally, this module sends the verification result of the trace ($\sigma \models \phi$) as either *true* or *false* to the analysis module.

The analysis module counts the number of total simulation traces as well as the number of simulation traces that satisfy the property. Using this information, the SIMVA-SoS tool uses a sequential probability ratio test (SPRT) [11] that statistically calculates the number of sufficient traces for a reliable verification result. Thus, if the number of traces is insufficient, this module calls the simulation module to request more simulations until it collects a sufficient number of samples. Otherwise, this module returns the probabilistic result in which the input model S satisfies the verification property ϕ. Two precision parameters (α , β) are used to check errors of accepting the probabilistic result of the SPRT algorithm [12]. Therefore, an error probability of the probabilistic result is bounded by a false positive probability α and a false negative probability β.

B. Slicing Concepts

The concept of the program slicing technique was first introduced by Weiser [13] to simplify programs based on some *slicing criteria*, which could include a specific variable v and a program point l (e.g., a line number in the program) that the user wants to investigate. The technique then generates a *program slice* that only contains statements that are relevant to this variable, relative to the program point.

Program slicing techniques can be categorized into three dimensions [14], *Forward/Backward* slice(or *Chopped slice* [15]), *Closure/Executable* slice and *Static/Dynamic* slice (or hybrid approaches [14]). A forward slice contains all statements that are affected by the slicing criteria, while a backward slice contains the statements that affect the slicing criteria. A closure slice is a closure of all the program statements that affect (in the case of backward slicing) or are affected by (in the case of forward slicing) the slicing criteria. These slices often produce subsets of the program that are non-executable; hence, closure slices cannot be used directly for generating executable programs. An executable slice is a subset of closure slices, which also require the slice to be executable. A static slice contains all possible execution paths, relative to the slicing criteria. A dynamic slice is based on a specific execution, which is to be analyzed; as such, we can construct the slice by *tracing* the execution path and recording any statements that are related to the slicing criteria, resulting in the dynamic slice producing a significantly smaller slice than that of a static slice.

III. An Illustrative Example

A. MCI-response SoS Scenario

In this paper, we use an MCI-response SoS model that has clear characteristics of an SoS, such as autonomy, diversity and connectivity [16]. The model assumes a scenario where a tsunami has occurred, thereby causing human casualties both on land and at sea. The model is composed of an MCI-response SoS manager and a Patient Transferring System (PTS). The SoS manager's goal is to rescue as many patients as possible from the various regions using the available PTSs. The SoS manager is trying to reduce the number of deaths by

```
 1 int reportStatus(Ambulance amb) {
 2   amb.status = 1;
 3   int distance_to_patient = new Random().nextInt();
 4   if (distance_to_patient < amb.max_distance) {
 5     amb.patients += 1;
 6     amb.status = 2;
 7   }
 8   System.out.println(amb.patients);
 9   distance_to_patient = 0;
10   return amb.status;
11 }
```

(a) The original example

```
 1 int reportStatus(Ambulance amb) {
 2   amb.status = 1;
 3   int distance_to_patient = new Random().nextInt();
 4   if (distance_to_patient < amb.max_distance) {
 5
 6     amb.status = 2;
 7
 8
 9
10   return amb.status;
11
```

(b) The example sliced by JavaSlicer

```
 1 int reportStatus(Ambulance amb) {
 2   amb.status = 1;
 3
 4
 5
 6
 7
 8
 9
10   return amb.status;
11 }
```

(c) The example sliced by ORBS

```
 1 int reportStatus(Ambulance amb) {
 2   amb.status = 1;
 3   int distance_to_patient = new Random().nextInt();
 4   if (distance_to_patient < amb.max_distance) {
 5
 6     amb.status = 2;
 7   }
 8
 9
10   return amb.status;
11 }
```

(d) The correct example slice

Fig. 1: Slicing the example SoS model

attending to the emergency patients first. However, each PTS prioritizes the closest patients, regardless of the urgency, due to the cost of moving the PTS. Therefore, the SoS manager assigns a PTS to the accident site. A patient on land is rescued by an ambulance, and a patient at sea is rescued by a helicopter. The PTS arrives at the accident site, sends a message to the SoS manager, and performs triage. First aid is provided to an injured patient in the PTS. The PTS then sends a request to a hospital and confirms the number of patients that can be accommodated in the hospital. At the same time, the PTS drives to the hospital.

B. Slicing the example

To introduce how these slicing techniques work, we have provided a small function from an example model in Fig. 1. The function in Fig. 1a has one Ambulance object as input. This object contains the current status of the vehicle, the number of patients that it carries, and the max distance that it is willing to travel to save a patient. The status indicates if the ambulance is searching for patients or if it is currently carrying patients. For this example we are only interested in the ambulance's status, and can thus specify the slicing criteria as the ambulance's status at the return statement on line 10. The random variable $distance\_to\_patient$ is an artificially random variable in this example to make a compact and complete example without describing a larger complex SoS.

This paper's algorithm is partly based on the *JavaSlicer* [5], which produces closure slices through dynamic backward slicing. This implies that the slice only contains the statements that affect the computation of $amb.status$ before line 10 for a specific input. Therefore at least one of the simulations has to provide an Ambulance object that satisfies $distance\_to\_patient < amb.max\_distance$ for the slices to include the correct trajectory as in Fig. 1b. These slices are computed using a dependence graph to include all of the statements that alter $amb.status$, including transitive control dependencies and data dependencies. The main disadvantage of JavaSlicer's approach is its inability to detect lines that are necessary to make a syntactically correct executable program. In Fig. 1b all relevant statements that affect the criteria are included, but lines that exclusively contain bracket delimiters are not included.

In the proposed technique, we also use a modified version of *observation-based* slicing (ORBS) — a language-agnostic slicing technique by Binkley et al. [6] to create executable slices. The ORBS technique is similar to dynamic slicing, but it uses an iterative reduction approach instead of creating dependence graphs. The technique utilizes iterative applications of deletion operators and *observes* whether the program slice behaves identically to the original program with respect to the slicing criteria. The observation is based on the slice being compilable and the *state trajectory* of the sliced program remaining identical to the original program. The state trajectory can be informally described as the sequence of states that are produced from the slicing criteria. If any of these conditions fail, the deletion is reverted, and the algorithm moves to a different program point. Given these properties, the resulting slice is always semantically equivalent to the program with respect to the slicing criteria.

However, the example program has the random value $distance\_to\_patient$, which causes the trajectory to vary between executions. The example program needs at least 12

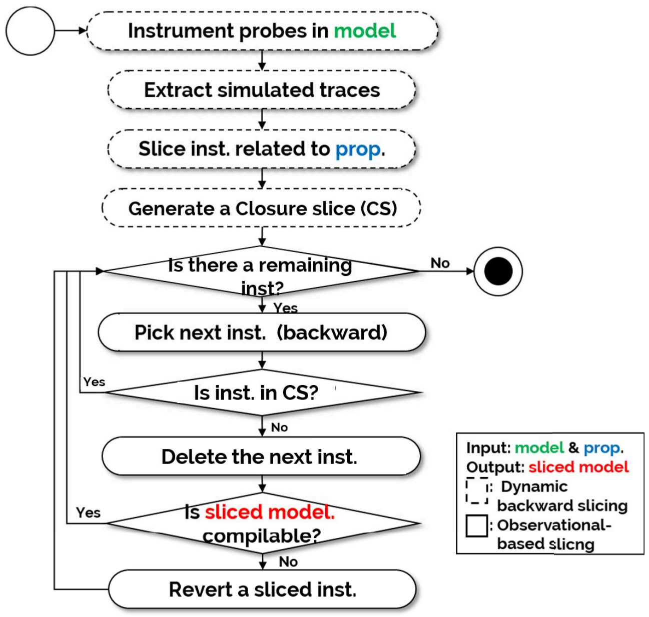

Instrument probes in model
Extract simulated traces
Slice inst. related to prop.
Generate a Closure slice (CS)
Is there a remaining inst?
No
Yes
Pick next inst. (backward)
Yes
Is inst. in CS?
No
Delete the next inst.
Yes
Is sliced model compilable?
No
Revert a sliced inst.
Input: model & prop.
Output: sliced model
: Dynamic backward slicing
: Observational-based slicng

Fig. 2: The overall approach to probabilistic SoS simulation model slicing

times of the deleting-observing-reverting process to check every program point. If the trajectories did not execute the *if* block, the ORBS technique would delete the *if* block and produce the incorrect slice shown in Fig. 1c. This slice produced by ORBS is executable but does not contain all of the same statements as the correct program slice in Fig. 1d. In essence, the dynamic slicing approach constructs a slice with an increasing amount of information for each iteration, while the ORBS technique iteratively reduces the slice, and thus might lose necessary information in the process when the behavior is probabilistic.

IV. Slicing Probabilistic SoS Simulation Models for SMC

This section introduces a probabilistic SoS simulation model slicing approach and its implementation in the statistical verification of SoSs. The proposed slicing approach utilizes dynamic backward slicing and the ORBS algorithm, which is described in Section III. The proposed approach is implemented as a slicing module in SIMVA-SoS, but could also be applied for general source code.

A. Procedure of Slicing Probabilistic SoS Simulation Model

The goal of the proposed approach is to generate an SoS simulation model slice that is based on the SoS verification properties or analysis criteria. Fig. 2 shows the overall approach of the SoS simulation model slicer, which includes two parts: 1) dynamic backward slicing (the upper parts, which are indicated by dotted lines) and 2) modified observation-based slicing (the lower parts, which are indicated by solid lines). After performing modified observation-based slicing

```
 1:  The dynamic slice for criterion [ChangedSoS.main:34:*]:
 2:  ChangedSoS.main:6 NEW java/util/ArrayList
 3:  ChangedSoS.main:6 DUP
 4:  ChangedSoS.main:6 INVOKESPECIAL java/util/ArrayList.<init>()V
 5:  ChangedSoS.main:6 ASTORE 1
 6:  ChangedSoS.main:10 ALOAD 1
 7:  ChangedSoS.main:10 INVOKEVIRTUAL java/util/ArrayList.add
                         (Ljava/lang/Object;)Z
 8:  ChangedSoS.main:11 ALOAD 1
 9:  ChangedSoS.main:11 INVOKEVIRTUAL java/util/ArrayList.add
                         (Ljava/lang/Object;)Z
10:  ChangedSoS.main:12 ALOAD 1
11:  ChangedSoS.main:12 INVOKEVIRTUAL java/util/ArrayList.add
                        (Ljava/lang/Object;)Z
```

Fig. 3: An intermediate output of the proposed approach

(mORBS) repeatedly, we can obtain executable probabilistic SoS simulation model slices as an output. Dynamic backward slicing provides an intermediate slice, which is used as the slicing criteria for the mORBS. The mORBS deletes the instructions (inst.) according to the intermediate slice information, not to the trajectory information.

Dynamic backward slicing is divided into four steps: inserting probes[3] into the probabilistic SoS simulation model, extracting the simulated traces, slicing the traces based on the specified properties, and generating a closure slice of the probabilistic SoS simulation model.

The proposed approach inserts probes into the probabilistic SoS simulation model to analyze the control and data dependency graphs as well as to see which instructions are executed when the model is running. Probe insertion in the probabilistic SoS simulation model is mainly performed at the Java bytecode level. When the probabilistic SoS simulation model with the probes is executed, JavaSlicer outputs a simulation trace in the form of a log. Backward slicing is performed on the extracted simulation traces. The slicing criteria of the simulation traces can be either variables or statements for model analysis, and verification properties for statistical verification. Slicing based on variables or statements follows a general backward slicing technique. Slicing based on verification properties involves analyzing the variables that are included in the verification properties as well as performing backward slicing. As a result of the slicing, an intermediate result (a closure slice) is generated (Fig. 3). The slicing criteria for the closure slice is written in the first line of Fig. 3. The example slicing criteria is on line 34 of the *main* function, which is included in the *ChangedSoS* class. The lines from line 2 to the last line of the example intermediate output include relevant Java bytecode instructions, including a class name, a function name, and a line number.

The result of the dynamic backward slicing (the closure slice) becomes an input for the next step. As described earlier in Section III, ORBS repeats the deleting-observing-reverting procedure. The difference between the original ORBS and mORBS is shown in the deleting and observing steps. Unlike ORBS, the closure slice guides mORBS on which statements to delete in the deleting step. In the observing step, mORBS

[3]Observing internal details of execution by inserting code snippets into a program

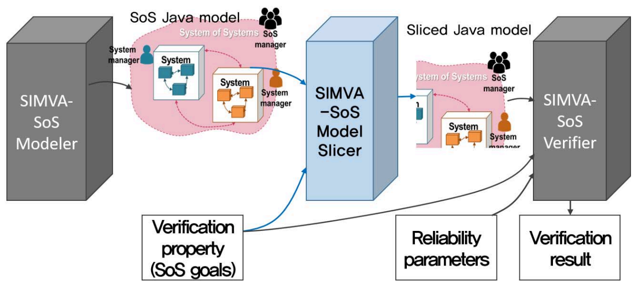


Fig. 4: The overall architecture of SIMVA-SoS

observes only the syntax errors in compilation instead of comparing trajectories of an SoS model. In this way, mORBS considers the random variables' dependencies and can still generate executable models.

In the mORBS part, the instructions are deleted one by one from the last instruction of the probabilistic SoS simulation model. However, if the instruction exists in the closure slice, the proposed approach skips the deletion of the instruction and deletes the next instruction. For example, after line 7 in the *ChangeSoS* class's *main* function is deleted, the next line to be deleted would be line 6. The proposed approach will detect that line 6 is contained in the closure slice, thus skipping line 6 and continuing to line 5.

The probabilistic SoS simulation model slice is compiled for execution with one instruction deleted. If compilation errors occur, the proposed approach reverts the deleted instruction. When the proposed approach compiles the probabilistic SoS simulation model, the compiled Java bytecode has lost some information, such as where curly-brackets ({ }) were written in the model. Thus, the closure slice does not have such information. If line 9 of the *main* function that is included in the *ChangedSoS* class contains a curly bracket and it is deleted by the approach, the SoS probabilistic simulation model without line 9 will invoke compilation errors. As a result, the approach reverts to line 9 and tries to delete the next line. This process repeats until there are no more instructions to delete. After completing all the steps of the slicing approach, an executable SoS simulation model slice is created with respect to the verification property.

B. Architecture of SIMVA-SoS including a Slicing Module

The overall approach of SIMVA-SoS is divided into three parts, as shown in Fig. 4: the modeler, the verifier, and the slicer. The modeler generates an SoS model based on the SoS meta-model; M2SoS [16], the verifier performs statistical model checking for the probabilistic SoS simulation model, and the model slicer performs model slicing with respect to the verification properties for efficient verification.

The probabilistic SoS simulation model slice generated by the slicing module can be used for analyzing and debugging purposes. Users can extract the parts that are related to specific variables, statements, or verification properties in which the user is interested. The slicing criteria given to the SIMVA-SoS model slicer include the package name, function name, line number, and variable name. Users can input the executable probabilistic SoS simulation model slice and other libraries/programs to SIMVA-SoS verifier. The executable probabilistic SoS simulation model slice is not automatically given to the SIMVA-SoS verifier due to dependency with other libraries. To improve verification efficiency and get accurate verification results, the SIMVA-SoS slicer and the SIMVA-SoS verifier should be given the same verification properties. If different verification properties are given to the slicer and verifier then the verifier will naturally provide an incorrect result.

V. Evaluation

We performed experiments on the MCI-response SoS simulation model to evaluate the proposed slicing technique. The following three research questions were answered:

- **RQ1**. How accurate is the proposed technique from the perspective of model similarity between the slice that is generated by the proposed technique and the slice that is generated by the static slicing technique?
- **RQ2**. Is the verification result of the original SoS simulation model the same as that of the sliced SoS simulation model?
- **RQ3**. How much time is saved when we perform statistical model checking on the sliced SoS simulation model?

We manually generated an SoS model slice following the static backward slicing algorithm. We compared the manually generated SoS model slice and the SoS model that was generated by our technique in RQ1 to see how similar the two models are. By answering RQ2, we can investigate the verification accuracy by comparing the verification result of the sliced model and that of the original model. The cost efficiency of the proposed slicing technique can be analyzed via RQ3. We compared the verification time of both the original and the sliced model.

A. Experimental Setup and Design

For RQ1, the SoS simulation model which contains the above MCI-response scenario was sliced to analyze which part of the system performed the rescue of patients on land. We measured the accuracy of the slicing technique by comparing the manually-generated MCI-response SoS simulation model with the MCI-response SoS simulation model slice. To perform the experiment for RQ2 and RQ3, we need the original model, the slicing criteria, the verification properties, and model slices. The SIMVA-SoS tool supports the verification of six properties:

1) The probability that the number of patients on land will be the same as the total expected number of patients is more than the expected probability (existence).
2) Until the end of the simulation, the probability of the resurrection of a dead patient on land is less than expected (absence).

3) The probability that the ambulance is always in the accident area by the end of the simulation is more than expected (universality).
4) The probability that the number of patients who are rescued on land after the designated time is greater than that the threshold is likely to exceed the expected probability (transient state).
5) The probability that the number of patients who are rescued from land over a long period of time until the end of simulation is greater than that the threshold is more than expected (steady state).
6) The probability that the remaining number of patients on land will exceed the threshold is maintained for at least 100-unit times before the end of the simulation (minimum duration).

For these six verification properties, we compared the results of the original model with the model slice in RQ2, and compared the verification times in RQ3.

We set the sample size up to 1,500 for the each verification property, as well as setting the α and β as 0.05 for checking *Type I error* and *Type II error* respectively. Similarly, the maximum number of steps for each sample was set to 200. We conducted the model verification using an Intel i7-7700 at 3.6GHz with 16GBs of memory on a 64-bit Windows 10 machine.

B. Experimental Results

RQ1. We compared the sliced model and the manually generated model slice, per the static slicing algorithm [13], line by line. Because ORBS is conducted on a line-by-line basis, a difference between the models can occur by the deletion of lines. We used the "compare" function of *Notepad++* to perform the comparison. The left model in Fig. 5 is the manually-generated model slice, and the right model is the SoS model slice that was generated by the proposed technique. Lines that were not in the model slice on the right when it was compared to the manually generated model on the left are colored in red. The formula for accuracy measure was as follows:

$$\frac{A \cap B}{A \cup B} \times 100, \quad \begin{array}{l} A: \text{manually generated model slice} \\ B: \text{automatically generated model slice} \end{array}$$

As a result, 96.55% of the lines were similar to the manually-generated model slice. The remaining 3.45% was caused by the characteristic of sampling execution. Although slicing the SoS model based on the simulation traces could improve the efficiency, it could also degrade to some extent in accuracy. For example, lines 18 to 22 in Fig. 5 show that the patients occur in double, with a one-in-ten thousand chance. With the number of patients determined, the simulation is executed in lines 23 and 24. Per the algorithm of static backward slicing, line 21 should not be deleted (left-hand side in Fig. 5). However, the SoS model slice that is generated by the proposed technique deletes line 23 because that line would be executed with a rare probability; therefore, it is not in the simulation trace.

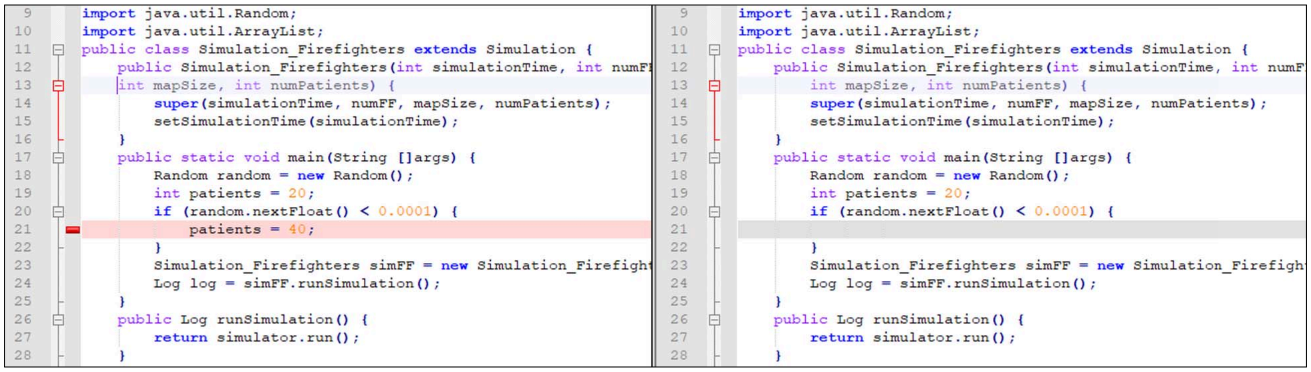
RQ2. When slicing for verification rather than slicing for analysis, it is necessary to confirm whether the verification result of the model slice is the same as the verification result of the original model to confirm the accuracy. After slicing the original model based on slicing criteria and verifying the original and sliced models on the six verification properties, the verification results of the model slices and the original model were compared. Fig. 6 demonstrates the results, according to the six verification properties; the maximum difference of the verification results was 2%. We think that the difference between the two verification results was due to the characteristics of statistical verification. Because the input model of the statistical model checker has uncertainty, the verification results may vary slightly depending on the simulation executions.

RQ3. Finally, we verified how much time efficiency can be achieved by slicing the model. We measured the verification time of the SoS simulation model slices and the original SoS simulation models, according to the six properties, while answering RQ2. Fig. 7 shows the verification time for two models for each verification property. As shown in the graph, the verification time was reduced by about half in all the verification properties. For instance, the most time-reduced property was the existence (56.45%), and the least was the transient state (47.59%). Because simulation is the most expensive aspect of SIMVA-SoS, we can significantly reduce simulation times by reducing the model's size. About 10% of the SoS model's size was reduced by the proposed approach, which was mainly related to simulation of rescuing patients at sea.

C. Limitations

Although the SIMVA-SoS model slicer module provides an automated analysis, one disadvantage is that the user has to manually input the model slice and the verification properties into the SIMVA-SoS verifier module. Another limitation is that we only compared two verification results of the original model and the model slice in RQ2, and simple comparison might not be enough to accurately reflect the slicing techniques. In addition, we were limited by some of the original techniques that we utilized. For example, the limitations of JavaSlicer can lower the accuracy of the model slice for the following reasons:

1) JavaSlicer cannot insert probes into some standard library classes, such as java.lang.String, java.lang.System, and java.lang.Object.
2) There are limitations to tracing multi-threaded applications correctly; JavaSlicer does not construct integrated data dependency graphs for different threads, and the subject model that was used in the experiment is not a multi-threaded application. This can be a problem when the tool is used in extended MCI-response SoS models or other domain SoS models that involve multi-threaded applications.
3) JavaSlicer is only available in the JDK 1.6 and 1.7 versions.



```
import java.util.Random;
import java.util.ArrayList;
public class Simulation_Firefighters extends Simulation {
    public Simulation_Firefighters(int simulationTime, int numFF
    int mapSize, int numPatients) {
        super(simulationTime, numFF, mapSize, numPatients);
        setSimulationTime(simulationTime);
    }
    public static void main(String []args) {
        Random random = new Random();
        int patients = 20;
        if (random.nextFloat() < 0.0001) {
            patients = 40;
        }
        Simulation_Firefighters simFF = new Simulation_Firefight
        Log log = simFF.runSimulation();
    }
    public Log runSimulation() {
        return simulator.run();
    }
```

```
import java.util.Random;
import java.util.ArrayList;
public class Simulation_Firefighters extends Simulation {
    public Simulation_Firefighters(int simulationTime, int numF
        int mapSize, int numPatients) {
        super(simulationTime, numFF, mapSize, numPatients);
        setSimulationTime(simulationTime);
    }
    public static void main(String []args) {
        Random random = new Random();
        int patients = 20;
        if (random.nextFloat() < 0.0001) {

        }
        Simulation_Firefighters simFF = new Simulation_Firefigh
        Log log = simFF.runSimulation();
    }
    public Log runSimulation() {
        return simulator.run();
    }
```

Fig. 5: The result of comparing the manually-generated model slice (L) and the model slice that was generated by the proposed technique (R)

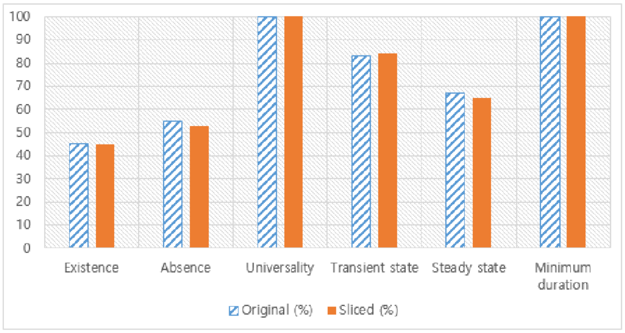


Fig. 6: Comparing verification results of the six verification properties between the original and the sliced models

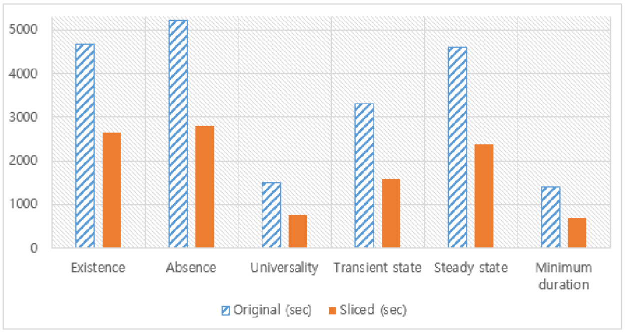


Fig. 7: Comparing the verification time of the six verification properties between the original and the sliced models

In SIMVA-SoS, we use Java as an intermediate representation between the SoS modeling language and bytecode. Thus our algorithm slices this general intermediate representation, instead of the specific higher-level modeling language. We have a plan to apply our approach to other executable SoS modeling languages.

VI. Related Work

Although SMC techniques generally scale better than other model checking techniques with an increase in the size of the state space [8], there remain performance issues that are related to a single simulation time and the repetitive number of samples generated.

Basu et al. [17] proposed an approach to efficiently apply SMC techniques to heterogeneous systems. This approach decreases the simulation time by executing not the entire system, but each application under a specific execution context respectively. This approach mainly focuses on generating the specific execution context that describes all interactions with other applications. However, this approach requires a preliminary stage for context generation and the execution time of the stage is directly affected by the number of component systems. By contrast, because our technique uses slicing technique, the performance is not affected by the number of component systems.

Jegourel et al. [18] presented the SMC platform, PLASMA, which reduces both simulation time and the number of samples for verification. PLASMA minimizes the number of states that a simulation trace contains by compiling the model and the properties into bytecode in advance and executing the bytecode on its own simulation and logic virtual machine. PLASMA also reduces the number of simulations by using importance sampling to frequently detect rare behaviors. The importance sampling tilts any probabilistic distributions of the model and artificially increases the probability of rare behaviors. However, a disadvantage to rare behaviors is that developers must manually optimize the parametric values of the tilted distribution with many simulations.

There are several related works on slicing in the context of the model checking and slicing probabilistic program [3], [19], [20]. Hatcliff et al. [19] proposed a deterministic slicing algorithm for reducing the model that is needed for the exhaustive model checking technique. By contrast, our work focuses on slicing models for statistical model checking.

Hur et al. [20] presented an algorithm for slicing probabilistic programs that focuses on extending traditional slicing techniques for programs that are written in a probabilistic programming language. The observing property of probabilistic program languages necessitates an extension of the traditional dependence graph to account for changes that are based on probabilistic observation. This slicing algorithm is therefore based on the language syntax of probabilistic programming languages; as such, it is not directly extensible to general-purpose languages, such as Java.

The SoS GaP Slicer by Song et al. [3] is a slicer that is made to specifically slice models based on the dynamic changes of SoS PRISM models. However, our proposed solution focuses on the probabilistic SoS model analysis based on variables, statements, and verification properties rather than changes to the model. Therefore, our solution is more general in its use of the features of slicing models that were written by using Java.

VII. Conclusion

In this paper, we proposed the SIMVA-SoS model slicer approach, including a SIMVA-SoS tool. Dynamic backward slicing and observation-based slicing were combined in the proposed technique. The generated model slice was formatted in an executable Java program. Applying slicing techniques to SoS simulation models is helpful in analyzing large, complex SoSs efficiently as well as for statistically verifying a portion of the system. Compared with the oracle model, the model slice that was generated by the proposed technique showed an accuracy of 96.55%. A comparison of the results of the model slice and the original model showed that the verification time was reduced by 47-56%, while it preserved the verification results. In the future, we plan to apply the proposed technique to other domains as well as MCI-response SoS. SIMVA-SoS can support verification of general SoS models if the input models are provided in Java. We also plan to propose a similarity metric for measuring the accuracy of the slicing technique for statistical model checking.

Acknowledgement

This research was supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. 2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System), and Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (2017M3C4A7066212).

References

[1] A. Legay, J. Quilbeuf, and F. Oquendo, “Verifying System-of-Systems with statistical model checking,” *ERCIM News*, vol. 103, 2015.

[2] D. Seo, D. Shin, Y.-M. Baek, J. Song, W. Yun, J. Kim, E. Jee, and D.-H. Bae, “Modeling and Verification for Different Types of System of Systems using PRISM,” in *Proceedings of the 4th International Workshop on Software Engineering for Systems-of-Systems*. ACM, 2016, pp. 12–18.

[3] J. Song, Y.-M. Baek, M. Jin, E. Jee, and D.-H. Bae, “SoS GaP Slicer: Slicing SoS Goal and PRISM Models for Change-Responsive Verification of SoS,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 546–551.

[4] M. Jin, D. Shin, and D.-H. Bae, “ABC+: Extended Action-Benefit-Cost Modeling with Knowledge-based Decision-making and Interaction Model for System of Systems Simulation,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 2018, pp. 1698–1701.

[5] C. Hammacher, K. Streit, S. Hack, and A. Zeller, “Profiling Java Programs for Parallelism,” in *Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE)*, May 2009, pp. 49–55.

[6] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, “ORBS: Language-independent Program Slicing,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635893

[7] A. Legay, B. Delahaye, and S. Bensalem, “Statistical Model Checking: An Overview,” in *International conference on runtime verification*. Springer, 2010, pp. 122–135.

[8] H. L. Younes, M. Kwiatkowska, G. Norman, and D. Parker, “Numerical vs. Statistical Probabilistic Model Checking,” *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 216–228, 2006.

[9] W. Whitt, “Continuous-time Markov Chains,” *Dept. of Industrial Engineering and Operations Research, Columbia University, New York*, 2006.

[10] V. Nimal, “Statistical Approaches for Probabilistic Model Checking,” Ph.D. dissertation, University of Oxford, 2010.

[11] A. Wald, “Sequential Tests of Statistical Hypotheses,” *The annals of mathematical statistics*, vol. 16, no. 2, pp. 117–186, 1945.

[12] Y. Kim and M. Kim, “Hybrid Statistical Model Checking Technique for Reliable Safety Critical Systems,” in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 51–60.

[13] M. Weiser, “Program Slicing,” in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE ’81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[14] G. A. Venkatesh, “The Semantic Approach to Program Slicing,” in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI ’91. New York, NY, USA: ACM, 1991, pp. 107–119. [Online]. Available: http://doi.acm.org/10.1145/113445.113455

[15] D. Jackson and E. J. Rollins, “A New Model of Program Dependences for Reverse Engineering,” in *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’94. New York, NY, USA: ACM, 1994, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/193173.195281

[16] Y.-M. Baek, J. Song, Y.-J. Shin, S. Park, and D.-H. Bae, “A meta-model for representing system-of-systems ontologies,” in *2018 IEEE/ACM 6th International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. IEEE, 2018, pp. 1–7.

[17] A. Basu, S. Bensalem, M. Bozga, B. Caillaud, B. Delahaye, and A. Legay, “Statistical Abstraction and Model-checking of Large Heterogeneous Systems,” in *Formal Techniques for Distributed Systems*. Springer, 2010, pp. 32–46.

[18] C. Jegourel, A. Legay, and S. Sedwards, “A Platform for High Performance Statistical Model Checking–PLASMA,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 498–503.

[19] J. Hatcliff, M. B. Dwyer, and H. Zheng, “Slicing Software for Model Construction,” *Higher-Order and Symbolic Computation*, vol. 13, no. 4, pp. 315–353, Dec. 2000. [Online]. Available: https://doi.org/10.1023/A:1026599015809

[20] C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel, “Slicing Probabilistic Programs,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594303